NISTIR 6171

Integration Strategies for the Reactive Scheduling

David Wyn Thomas
Guest Researcher

Manufacturing Systems Integration
Division
MSI Research Institute
Loughborough University

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards
and Technology
Gaithersburg, MD 20899-0001

NIST

NISTIR 6171

Integration Strategies for the Reactive Scheduling

David Wyn Thomas
Guest Researcher

Manufacturing Systems Integration
Division
MSI Research Institute
Loughborough University

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards
and Technology
Gaithersburg, MD 20899-0001

June 1998

U.S. DEPARTMENT OF COMMERCE
William M. Daley, Secretary

TECHNOLOGY ADMINISTRATION
Gary R. Bachula, Acting Under Secretary
for Technology

NATIONAL INSTITUTE OF STANDARDS
AND TECHNOLOGY
Raymond G. Kammer, Director

Integration Strategies for the Reactive Scheduling System

David Wyn Thomas
Guest Researcher, Manufacturing Systems Integration Division
MSI Research Institute, Loughborough University

Abstract

Shop Floor operations can be optimized by scheduling applications that not only create an initial schedule but also allow feedback from data collection systems to enable re-scheduling. Interfaces required to integrate the various commercial applications within such a reactive system are being developed at the National Institute of Standards and Technology (NIST). This report describes how standards-based, distributed object technology has been used to implement some of those interfaces.

1. Introduction to the Reactive Scheduling System.

NIST has been developing a Reactive Scheduling System (RSS), which is capable of modifying the existing schedule based on feedback from the shop floor, for a number of number years. The functional components that have been identified within the RSS are shown in Figure 1. These high level components may themselves be made up of sub-components. Information models to describe the status of the shop floor in terms a scheduler can understand have been defined previously[1].

Prior to the work covered in this document all integration and inter-component communication was implemented using files. The Scheduler component produced a description of a schedule from new order information, routings, and the current state of the shop floor. This was stored in a file that the Dispatcher then repackaged into chronological work orders, called jobs, for each available resource. A job specifies a machine resource to execute a production step on a group of parts, called a load. The Shop Floor simulator read this file as input and simulated the execution of the jobs to produce the loads. Status reports on the progress of loads and state of resources were stored in a file that the Status Manager periodically read. No detailed specification or implementation of the System Manager had been done. The implementation was very static and environment-dependent. The components were tightly coupled both to each other and to the environment in which they were located. Both the file location and the host machine on which a component executed had to be explicitly specified by any other component that wanted to communicate with it. If a component's physical location or host machine changed, any component that interacted with it would also need to be modified to maintain communication.

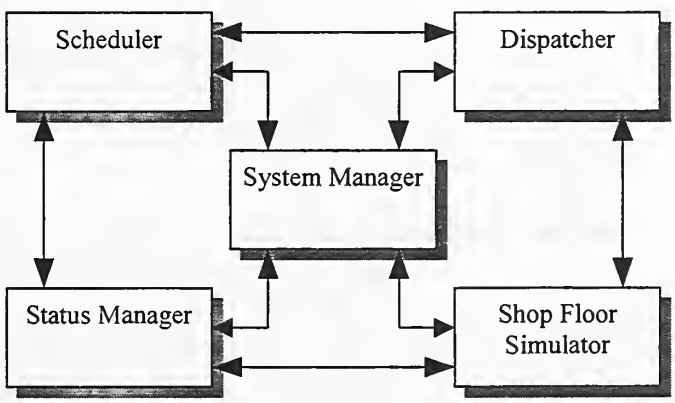

Figure 1: Interacting components within the RSS.

Information was transferred between components via text strings that were written to and read from files. An example text string passed between the Shop Floor Simulation and the Status Manager is shown below.

change_resource,"00:30:18","Fix_R", current_state, "busy", current_load,"L01Or01"

Relaying information between communicating components via text strings required the components to parse the strings and extract the information contained in them.

This document describes how distributed object technology has been used to replace the file-based implementation to produce a loosely coupled component system.

2. Introduction of Distributed Object Technology Communication Mechanisms

Object-oriented design and analysis methodologies[2] can be applied to the Reactive Scheduling System. The components shown in Figure 1 can be mapped to objects, and the messages that are passed between the components can be mapped to methods and attributes belonging to the objects. Using the object-oriented approach facilitates the separation of the interfaces from a component and implementation of that component. The well-defined interface is publicly available to all other components within the system, while the implementation remains encapsulated within the component.

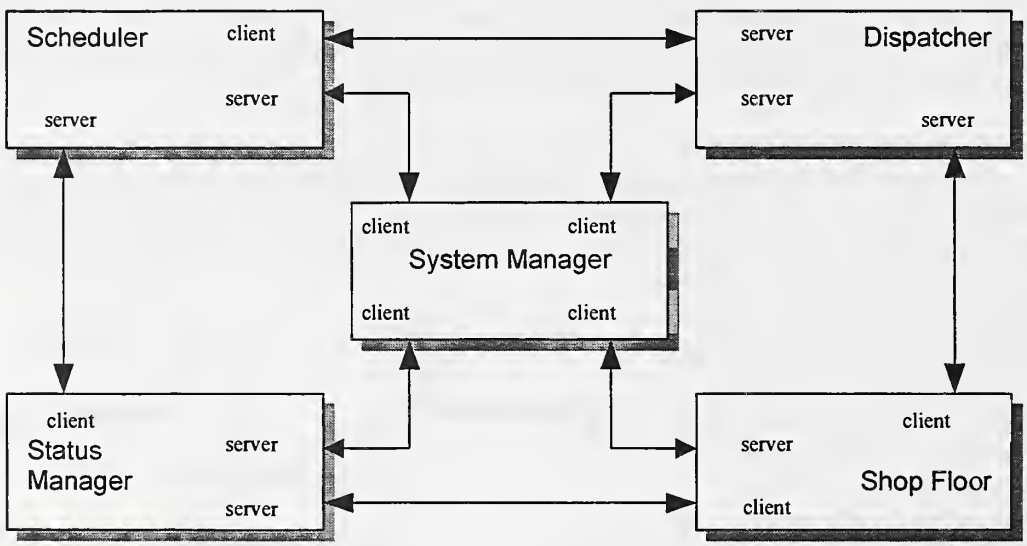

Figure 2: Client-Sever relationships of components

Distributed Object Technology extends object-oriented programming to enable objects that are physically remote from each other to interact. Communicating distributed objects may reside in heterogeneous environments in terms of host computer, operating systems, or networks. A change of implementation (e.g., using a different shop floor simulator) will require alterations only to the internals of the component. No alterations to the well-defined interface are necessary. Other communicating components will, therefore, be unaware that the internal implementation of any particular component has been changed.

To achieve this level of flexibility, a definition of the role each component performs with respect to the other entities within the system is required. Figure 2 outlines the dominant communication relationships

between the components in terms of the client-server architecture[3]. A client makes calls (requests) on a server. Each component can be either client or server to other components in the system, or both. Figure 2 shows that the Status Manager is a client of the Scheduler, and a server for the Shop Floor and the System Manager.

The components within the Reactive Scheduling System run as processes on remote host machines. Traditionally, remote Inter-Process Communication (IPC) has been accomplished using files or a low-level mechanism such as a socket interface. However these mechanisms for remote IPC tend to produce processes that are tightly bound to each other. A change in one process will often require subsequent changes in the remote communicating processes.

The introduction of a broker mechanism[4] into the system removes the distribution details from the individual components and places them in an external request broker component. This enables loose coupling both to other components and the host environments. The Common Object Request Broker Architecture (CORBA)[5] specification enables components to communicate to each other using distributed object technology. We have implemented a CORBA-compliant, distributed-object, system using the Orbix® * product from Iona Technologies[6].

Figure 2 shows the predominant information and control relationships among the manufacturing components that make up the RSS. However. a server may at times wish to notify a client

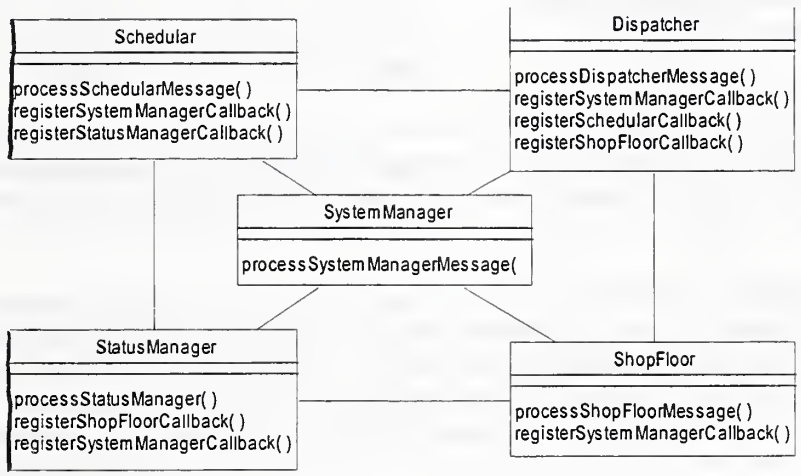

Figure 3: Class diagram of the RSS components.

asynchronously that an event has occurred. A distributed callback mechanism has been used to enable this behavior† that executes a method within the client component when such an event occurs in the server.

From the definitions of the components and their relationships to one another, a class diagram of the Reactive Scheduling System can be constructed. The Unified Modeling Language (UML)‡, an object-oriented modeling language has been used for this purpose.

† A distributed callback mechanism is a simple approach to enable bi-directional asynchronous client-server (peer-to-peer) behavior. Details of the Orbix implementation can be found in the OrbixWeb Programming Guide and also on the Iona web pages (http://www.iona.com). Iona's OrbixTalk is an implementation of the CORBA Event service.

The UML class diagram of the system components is shown in Figure 3. An iterative approach to the introduction of distributed object technology has been used within the Reactive Scheduling System. The first implementation replaces the ASCII text files with a simple CORBA interface. The messages contained within the text strings now pass across this interface. Currently, no information as to the type of data passing from one component to another is explicitly held in the interfaces. However, an iterative approach enables quick development of the CORBA wrappers with minimum alteration to the application code that has already been written.

One application-oriented method that appears in several of the classes shown in Figure 3 is *processXXXMessage*, which is used to pass the text strings from a client to a server component. For example the Shop Floor (client) would connect to the Status Manager (server) and then call the *processStatusManagerMessage* method, with a text string containing the relevant status information as a parameter of the method. This is shown below,

```
void processStatusManagerMessage(in
        string MessageString);
```

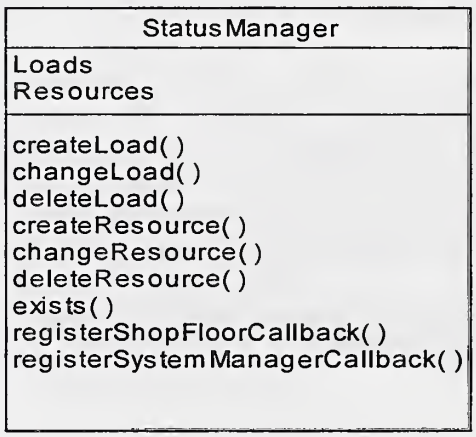

Figure 4: Richer Status Manager Class

The whole Reactive Scheduling System has been specified at this level. The components within the system that have been implemented with the interfaces described above are detailed in the section on *Implemented Components* below. A richer object-oriented interface for the components is required, where the type of data passing across the interface is specified explicitly. This work is continuing at present at NIST[7].

An example class diagram containing a richer interface for the Status Manager component is shown in Figure 4. It demonstrates the level of interface definition that will need to be defined for all of the components. The single *processStatusManagerMessage* method of the first design iteration has been replaced by several methods. A method is called depending on the status information that is being sent from the Shop Floor. This will reduce the degree of parsing performed in the component itself. The semantic content of the text string messages will be defined by an object interface, comprised of methods with parameters.

Below we show how a change_resource status message contained in a text string in the simple Status Manager interface will be replaced by a call to a changeResource method call on the StatusManager object. It contains four parameters, timestamp, resource name, current state and current load.

```
change_resource,"00:30:18","Fix_R",current_state,"busy",current_load,"L01Or01"
```

Replaced By:

```
void changeResource ( timestamp, resource, state, load );
```

‡ The Unified Modeling Language is a language for specifying, visualizing, and constructing the artifacts of software systems, as well as for business modeling. The UML represents a collection of "best engineering practices" that have proven successful in the modeling of large and complex systems (http://www.rational.com). It is based upon the Booch, Object Modeling Technology, and Jacobson methodologies.

The *registerCallback* methods, shown in the class diagram in Figure 4, are used to create the distributed callback mechanisms introduced above. These enable servers to call methods on objects located in clients. Tools are available that can be used to generate CORBA Interface Definition Language (IDL)-compliant interfaces for each of the components of the Reactive Scheduling System, from the UML class diagrams.

3. Developing a CORBA Wrapper for the Components

The IDL definitions of the interfaces of the components of the Reactive Scheduling System are converted into code "stubs" by an IDL compiler. The code stubs are used in implementation of the component wrappers. A wrapper enables applications that do not have CORBA interfaces to communicate with other distributed components that have CORBA-compliant interfaces. The output of the IDL compiler will be dependent on the programming language in which the wrapper is to be written. This could be C++, Java, or Smalltalk.

Figure 5 shows the files generated by the IDL compiler during the wrapper development from the UML class diagrams to the C++ source code. The Orbix IDL to C++ compiler generates two sets of files.

The implementation stubs contain the application-specific CORBA methods that call the underlying Orbix® libraries. The developer should not amend these.

The generated implementation skeletons contain an outline of the class and methods for an object that will be implemented within the server. The developer completes this class by writing the code that will be executed when a client makes a call on an object of this type.

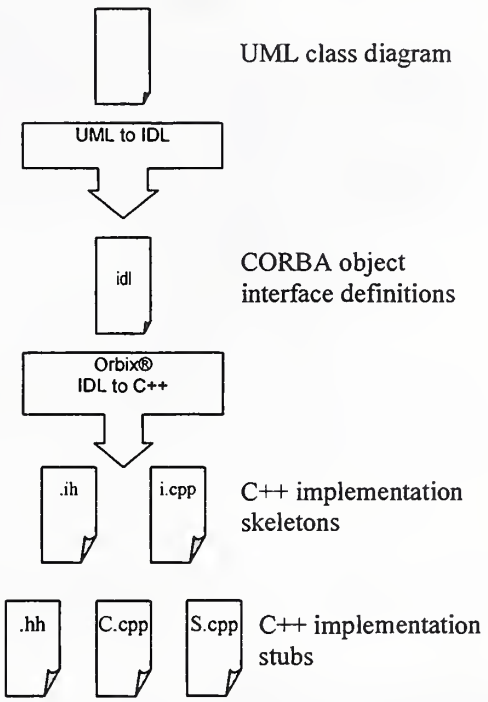

Figure 5: Files produced for wrapper development

4. Implemented Components

This section describes the work that has been completed within the initial implementation of the CORBA-based Reactive Scheduling System (RSS). The components for which a CORBA interface has been implemented are the Status Manager, the Shop Floor Simulator, and the Dispatcher. An implementation diagram of these components is shown in Figure 6.

The rest of this section outlines, in greater detail, the development of each of the current implementations of these components.

4.1 Status Manger

The current implementation of the Status Manager component has been developed in-house at NIST using Visual Basic® . State information arriving at the Status Manager from the Shop Floor is stored in

a database via an Open Database Connectivity (ODBC) link. To provide a CORBA interface to this component a CORBA/ActiveX gateway was required to enable ActiveX® objects in the Visual Basic® Status Manager to be accessible to remote CORBA client processes. To facilitate this, the IDL compiler produces a binary file containing all of the CORBA/ActiveX translation functionality. The produced binary is pasted onto a Visual Basic® form in the server process to provide access to this functionality.

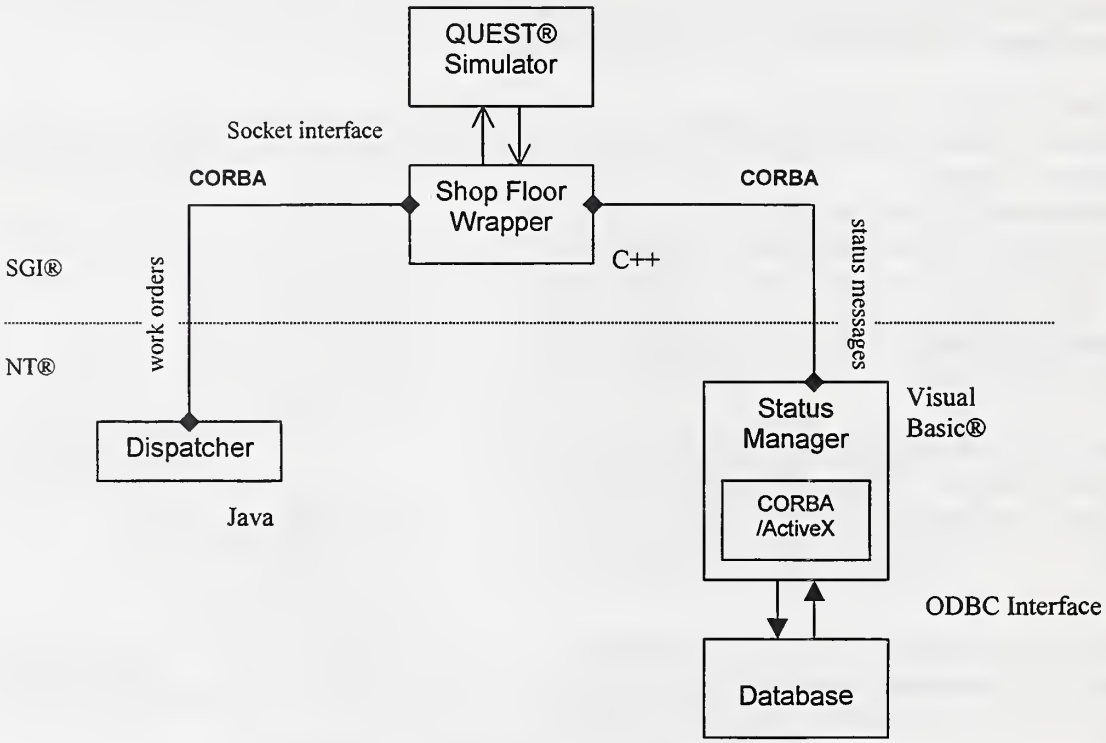

Figure 6: RSS Implementation Diagram

4.2 Shop Floor Simulator

The current Shop Floor Simulator implementation uses the Deneb Robotics Quest® simulation tool running under IRIX5.3® on a Silicon Graphics Computer® . The CORBA wrapper that connects the simulator with the rest of the components in the Reactive Scheduling System is implemented in C++ and compiled with the native SGI® compiler. The Shop Floor wrapper communicates with Deneb Robotics Quest® via a socket interface with the wrapper as the server and Deneb Robotics Quest as the client. Deneb Robotics Quest® requires a special message format for socket communications.

Byte	Contents:
1-5	Length of message. (ASCII Text)
6-N	Actual message.

The first 5 bytes read from the socket contain the length of the remaining message, the actual message the user wishes to pass over the socket. The message length is a right justified character string with leading spaces.

The wrapper executes an event handler that dispatches incoming events from both Deneb Robotics Quest (through the socket interface) and from the other remote communicating components within the Reactive Scheduling System (through the CORBA interface).

4.3 Dispatcher

The current implementation of the Dispatcher was developed in-house at NIST, and written in Java. It is presently a Java application as opposed to an applet and therefore cannot be accessed through an Internet browser. The Iona OrbixWeb® IDL to Java compiler, has been used to create the CORBA stubs and skeleton files.

5. Further Work

Development of the internal implementation of all of the components within the Reactive Scheduling System is continuing at NIST. The next step is to allow new orders to enter the system, delays in the completion of tasks, and the breakdowns of machine. Once this is completed, a new component will be added called the monitor. Its main functions will be to predict the impact of these types of events on the current schedule, and to determine when a new schedule is needed. Some type of Ai-based or agent system is envisioned.

Work on the definition of the interfaces between the components will continue in three areas:

- Complete the initial iteration of a CORBA-based distributed object Reactive Scheduling System. The Scheduler and System Manager components are yet to be implemented.
- Define and develop a richer CORBA interface for each of the components within the Reactive Scheduling System.
- Replace existing applications with other s that have the same functionality.

These activities will require further investigation into the functional role of each of the components and the communication required between them.

* DISCLAIMER:
Certain commercial software and hardware products are identified in this paper. This does not imply approval or endorsement by NIST, nor does it imply that the identified products are necessarily the best available for the purpose.

6. References

[1] F. Riddick, and A. Loreau, "Models for Integrating Scheduling and Shop Floor Data Collecting Systems", Proceedings of IASTED MIC 97, Innsbruck, Austria, 1997.

[2] G. Booch, "*Object-Oriented Analysis and Design*", Addison-Wesley, London, England, 1994.

[3] R. Orfali, D. Harkey, and J. Edwards, *"Essential Client / Server Survival Guide*", Van Nostrand Reinhold, New York, New York, 1994.

[4] R. Orfali, D. Harkey, and J. Edwards, "*The Essential Distributed Objects Survival Guide"*, J. Wiley & Sons, New York, 1996.

[5] Object Management Group, "The Common Object Request Broker: Architecture and Specification - Revision 2.0", http://www.omg.org, 1995

[6] IONA Technologies Ltd., "The Orbix Architecture", IONA Technologies Ltd, Dublin, Ireland, 1995.

[7] C. Lecapitaine, F. Riddick, and A. Jones: "IMES II - Production Management Standards: Requirements Analysis for Shop Floor Status," NISTIR 6123, National Institute of Standards and Technology, Gaithersburg, MD, 1998.